🔢 From Univariate to Multiple Linear Regression

📌 Univariate Linear Regression Recap:

- Model:
  fw,b(x)=wx+b
- One feature (e.g., house size) used to predict the target (e.g., price).

🧠 Introduction to Multiple Features

🏡 New Scenario:

We now consider **multiple features** to predict housing price:

- x_1: Size (sq. ft)
- x_2: Number of bedrooms
- x_3: Number of floors
- x_4: Age of house (in years)

This gives us **more information** for making better predictions.

🔤 New Notations

Term	Meaning
x_j	The **j-th feature** (e.g., size, bedrooms, etc.)
n	Total number of features
$x^{(i)}$	Feature vector of the **i-th training example** (a row vector)
$x_j^{(i)}$	Value of the j-th feature for the i-th example
$\vec{x}^{(i)}$	Vector notation to emphasize it's a list, not a scalar

📝 Example:

- $x^{(2)} = [1416, 3, 2, 40]$
- $x_3^{(2)} = 2$ (3rd feature = 2 floors)

🧾 Multiple Linear Regression Model

Expanded Form:

$$f_{w,b}(x) = w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4 + b$$

Example Model (housing price in $1000s):

$$f_{w,b}(x) = 0.1x_1 + 4x_2 + 10x_3 - 2x_4 + 80$$

Interpretation of Coefficients:

- `b = 80`: base price is $80,000
- `0.1`: +$100 per extra square foot
- `4`: +$4000 per bedroom
- `10`: +$10,000 per floor
- `-2`: -$2000 per year older

📦 General Form with `n` Features

$$f_{w,b}(x) = \sum_{j=1}^{n} w_j x_j + b$$

🧮 Vector Notation

To make the math cleaner, use vectors:

- Let:

$$\vec{w} = [w_1, w_2, ..., w_n] \quad \text{(parameter vector)}$$

$$\vec{x} = [x_1, x_2, ..., x_n] \quad \text{(feature vector)}$$

- `b`: remains a scalar

🔗 Dot Product (Inner Product)

🔗 Dot Product (Inner Product)

$$\vec{w} \cdot \vec{x} = w_1x_1 + w_2x_2 + ... + w_nx_n$$

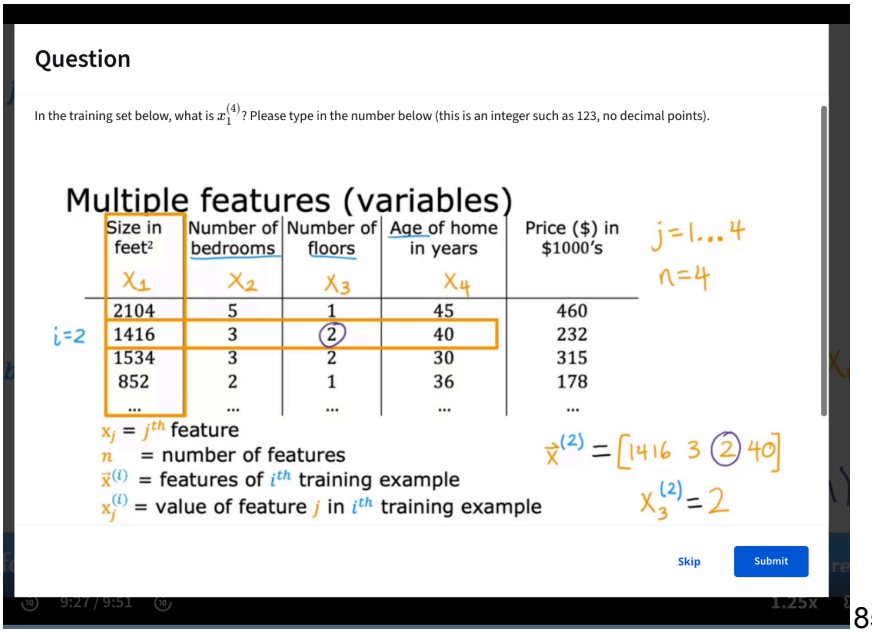
Final Model (Compact Form):

$$f_{w,b}(x) = \vec{w} \cdot \vec{x} + b$$

⚠️ Dot product only works when both `w` and `x` are of the **same length** (same number of features).

📌 Terminology

Term	Meaning
Univariate Linear Regression	Regression with a single feature
Multiple Linear Regression	Regression with multiple features
Multivariate Regression	A different concept (not covered here); often refers to predicting **multiple outputs**, not using multiple inputs



852 ans

🧠 What is Vectorization?

> **Vectorization** is a method of rewriting code (especially loops and repetitive operations) to leverage **matrix and vector operations**.

📈 Why Vectorization Matters in Machine Learning?

- ✅ **Shorter Code** → Easier to read, write, and maintain.
- 🚀 **Faster Execution** → Leverages modern linear algebra libraries and hardware acceleration (CPU/GPU).
- 🧮 Used extensively in machine learning for **efficient computation**.

🧪 Example: Without and With Vectorization

🔢 Setup:

Let's say:

- $\vec{w} = [w_1, w_2, w_3]$
- $\vec{x} = [x_1, x_2, x_3]$
- b is a scalar
- $n = 3$

🔸 Implementation without vectorization (manual multiplication):

python
CopyEdit

```
f = w[0] * x[0] + w[1] * x[1] + w[2] * x[2] + b
```

👎 Not scalable: what if `n = 100,000`?

🔸 Loop-based (still not vectorized):

python
CopyEdit

```
f = 0
for j in range(n):    # j = 0 to n-1
    f += w[j] * x[j]
f += b
```

⚠️ Still sequential and not fast when `n` is very large.

✅ Vectorized Implementation (Best Practice):

✨ Math:

$$f_{\vec{w},b}(x) = \vec{w} \cdot \vec{x} + b$$

✨ Code (NumPy):

```
import numpy as np
```

```
f = np.dot(w, x) + b
```

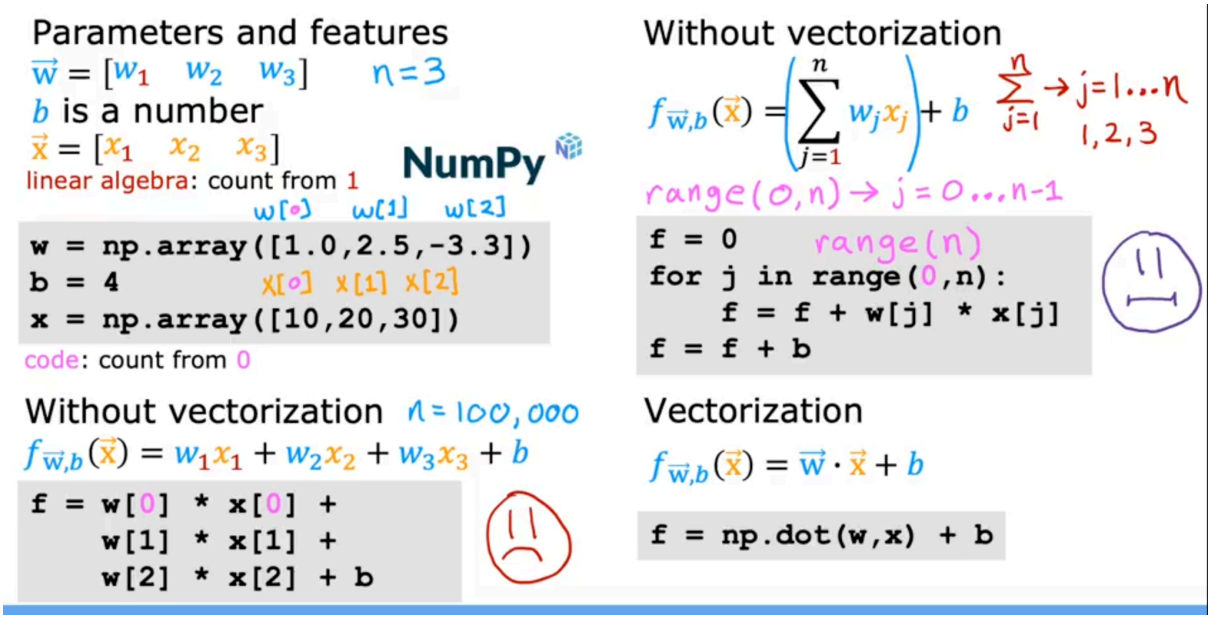
- `np.dot(w, x)` performs the **dot product**.
- Only **one line** does the full computation.

⚡ Why is Vectorization So Fast?

1. **Parallel Execution**
    - NumPy uses optimized **C/C++ backends**.
    - Operations are **executed in parallel** across CPU or **GPU** cores.
2. **No Python Overhead**
    - For loops in Python are slow due to interpreter overhead.
    - Vectorized code executes in **compiled code** under the hood.

💡 Recap: Benefits of Vectorization

Benefit	Description
📉 Shorter Code	One-liner instead of multiple lines
🏃 Faster Code	Uses efficient C/Fortran implementations and parallelism
📦 Scalable	Handles large `n` effortlessly
📚 Clearer	Easier to read and debug



⚙️ How Vectorization Works (Behind the Scenes)

🧠 Sequential vs Parallel Computation

🔁 Without Vectorization (For Loop)

- Executes operations **step-by-step**, **one after another**.

```
for j in range(16):
    result[j] = w[j] * x[j]
```

- Time steps:
  - t0t_0t0: compute w0×x0w_0 \times x_0w0×x0
  - t1t_1t1: compute w1×x1w_1 \times x_1w1×x1
  - …
  - t15t_{15}t15: compute w15×x15w_{15} \times x_{15}w15×x15

🔻 **Slow for large feature sets** (like `n = 10,000`).

⚡ **With Vectorization (Parallel Processing)**

\

```
result = np.dot(w, x)
```

- Uses **specialized hardware** (CPU/GPU)
- Computes all 16 multiplications **simultaneously**
- Then adds the 16 results efficiently with **parallel summation**

🔺 **Faster**, especially for large n.

🧮 Use Case: Parameter Updates in Gradient Descent

🚨 Goal:

Update weights wjw_jwj using gradient descent:

wj:=wj−α⋅djw_j := w_j - \alpha \cdot d_jwj:=wj−α⋅dj

Where:

- α\alphaα: learning rate (e.g., 0.1)
- djd_jdj: derivative of cost w.r.t wjw_jwj

❌ Without Vectorization

```
for j in range(16):
    w[j] = w[j] - 0.1 * d[j]
```

- Updates each weight **individually** (slower)

✅ With Vectorization

```
w = w - 0.1 * d
```

- **All 16 updates** done in one line
- Leverages **NumPy array broadcasting**
- Runs faster using **parallel computation**

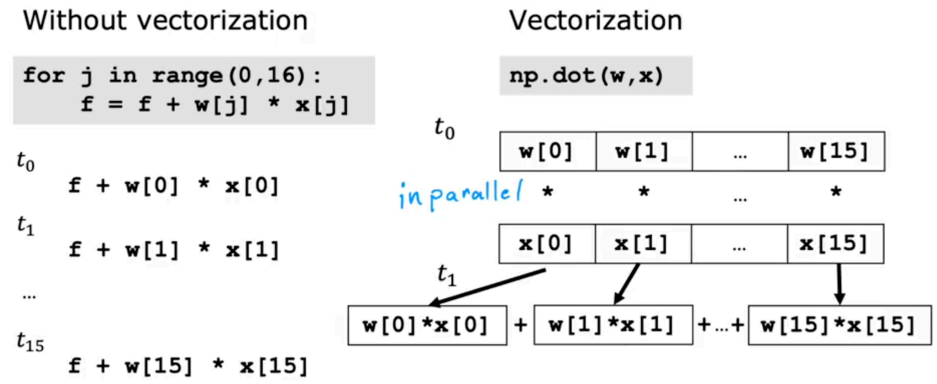
🔍 Practical Example in NumPy

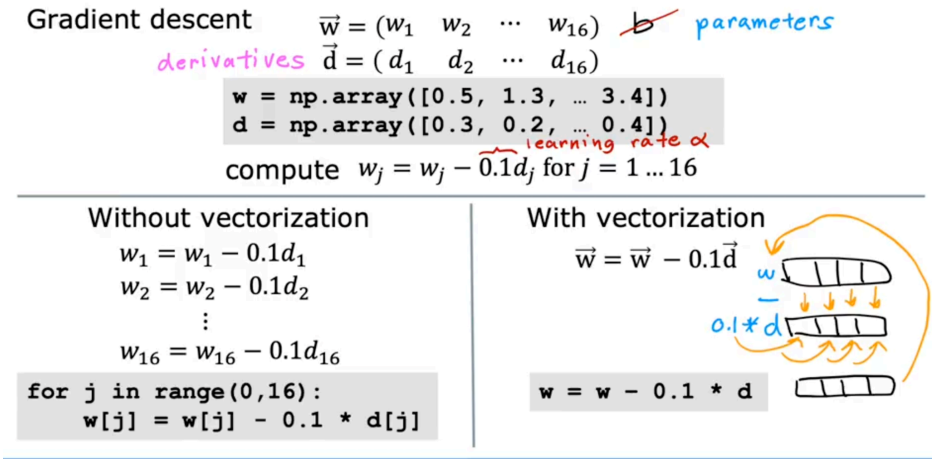
```
import numpy as np

# Parameters
w = np.array([w1, w2, ..., w16])
d = np.array([d1, d2, ..., d16])
alpha = 0.1

# Vectorized update
w = w - alpha * d
```

🚀 Real Impact

Situation	Speed Gain
Small n (e.g., 16 features)	Slight
Large n (e.g., 10,000+ features)	Huge
Large training dataset (m large)	Critical

⚠️ Vectorization can make the **difference between seconds and hours** in ML training!





🔁 Gradient Descent with Multiple Features (Vectorized Form)

📌 Recap: Multiple Linear Regression in Vector Form

- **Model:**

$$f_{\vec{w},b}(x) = \vec{w} \cdot \vec{x} + b$$

  - $\vec{w} = [w_1, w_2, ..., w_n]$: parameter vector (length = number of features)
  - $\vec{x} = [x_1, x_2, ..., x_n]$: feature vector
  - b: scalar bias term

🎯 Cost Function (Vectorized Notation)

- Let $J(\vec{w}, b)$ be the cost function (Mean Squared Error):

$$J(\vec{w}, b) = \frac{1}{2m} \sum_{i=1}^{m} (f_{\vec{w},b}(x^{(i)}) - y^{(i)})^2$$

  - m: number of training examples
  - Each $x^{(i)}$ is a **vector** of features

🧮 Gradient Descent: Updating Parameters

📉 For One Feature:

- Update rules:

$$w := w - \alpha \cdot \frac{1}{m} \sum_{i=1}^{m} (f_w(x^{(i)}) - y^{(i)}) \cdot x^{(i)}$$

$$b := b - \alpha \cdot \frac{1}{m} \sum_{i=1}^{m} (f_w(x^{(i)}) - y^{(i)})$$

📊 For Multiple Features:

- For each w_j (from 1 to n):

$$w_j := w_j - \alpha \cdot \frac{1}{m} \sum_{i=1}^{m} (f_{\vec{w},b}(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)}$$

- Update b as before:

$$b := b - \alpha \cdot \frac{1}{m} \sum_{i=1}^{m} (f_{\vec{w},b}(x^{(i)}) - y^{(i)})$$

⚡ Vectorized Gradient Descent in Code (using NumPy)

```
# X: shape (m, n) -> each row is a training example
# y: shape (m, ) -> actual outputs
```

```
# w: shape (n, ) -> parameters
# b: scalar
# alpha: learning rate

# Prediction
f = np.dot(X, w) + b             # shape (m, )

# Error term
error = f - y                    # shape (m, )

# Gradients
dj_dw = (1/m) * np.dot(X.T, error)  # shape (n, )
dj_db = (1/m) * np.sum(error)       # scalar

# Update
w = w - alpha * dj_dw
b = b - alpha * dj_db
```

🧠 Everything is done **efficiently in parallel**, thanks to vectorization.

📌 Summary of Vectorized Gradient Descent

Component	Description
`np.dot(X, w)`	Matrix-vector multiplication: all predictions at once
`np.dot(X.T, error)`	Efficient computation of all wjw_jwj gradients
Vectorization	Speeds up large-scale training and simplifies implementation

---

📍 Aside: The Normal Equation

🧾 What is it?

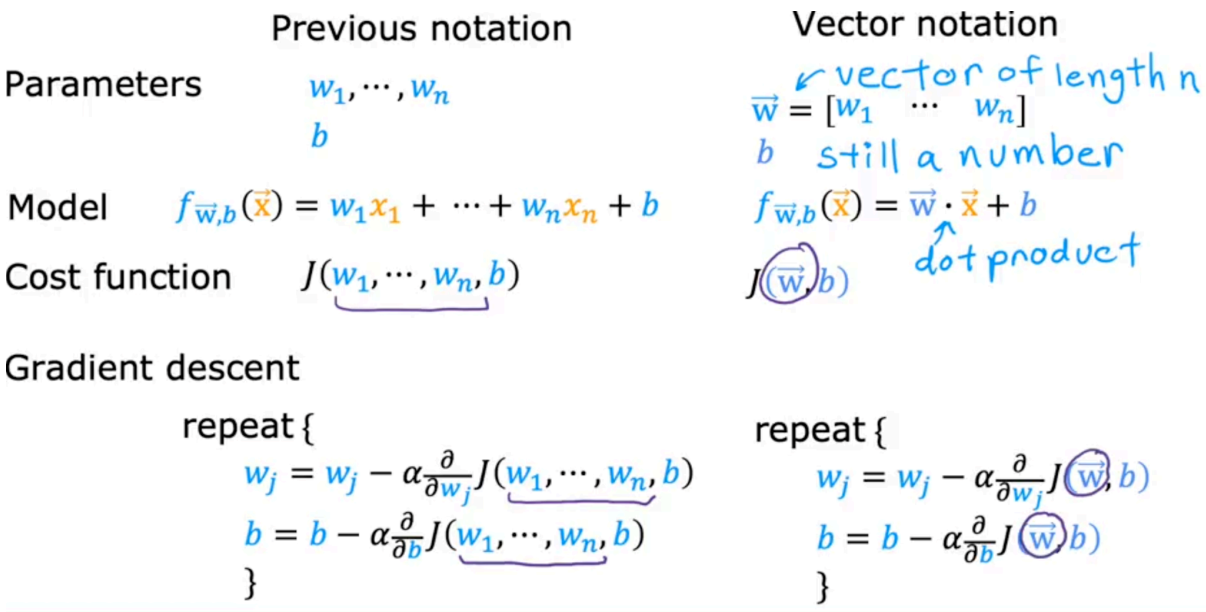
- An **analytical** solution to linear regression:

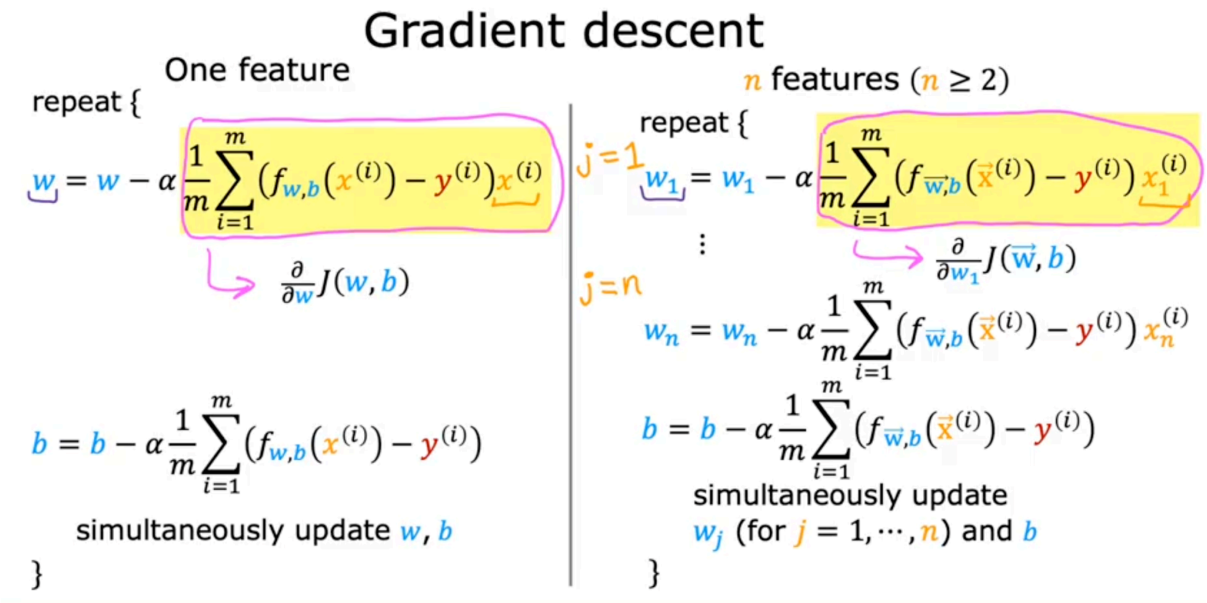
$$\vec{w} = (X^T X)^{-1} X^T y$$

🧩 Characteristics:

Feature	Description
✅ No iterations	Directly solves for w⃗\vec{w}w and bbb
❌ Slow for large n	Matrix inversion is computationally expensive
❌ Not generalizable	Only works for linear regression, **not** for logistic regression, neural networks, etc.

📝 *Most practitioners don't use it directly, but some ML libraries may use it under the hood.*





1 / 1 point

2. Which of the following are potential benefits of vectorization? Please choose the best option.

- ○ It makes your code run faster
- ○ It can make your code shorter
- ○ It allows your code to run more easily on parallel compute hardware
- ◉ All of the above

Correct
Correct! All of these are benefits of vectorization!

3. True/False? To make gradient descent converge about twice as fast, a technique that almost always works is to double the learning rate $alpha$. 1 / 1 point

- ◉ False
- ○ True

Correct
Doubling the learning rate may result in a learning rate that is too large, and cause gradient descent to fail to find the optimal values for the parameters w and b.

🔍 Problem Setup

You're trying to **predict house prices** using two features:

- **x_1 = size of the house (sq. feet)** → ranges from **300 to 2000**
- **x_2 = number of bedrooms** → ranges from **0 to 5**

That's a **big range difference** between x_1 and x_2.

🤔 Why Is This a Problem?

If features have very **different scales**, then:

- **Large feature values** (like size in sq. feet) will **dominate the cost function's sensitivity**.
- This leads to **elongated contours** (think of stretched ovals) in the cost function's landscape.

As a result:

- Gradient descent will take **zig-zag paths**, bouncing slowly down the valley.
- It **takes much longer to converge** to the optimal values of weights (w_1, w_2).

🧠 Example to Understand

Let's say you have a house:

- Size: **2000 sq. feet**
- Bedrooms: **5**
- Price: **$500,000 (500k)**

Case 1: Parameters → $w_1 = 50, w_2 = 0.1, b = 50$

Predicted price:

yaml
CopyEdit

```
= 50 × 2000 + 0.1 × 5 + 50
= 100,000 + 0.5 + 50 ≈ 100,100.5k → Way too high!
```

→ **Bad prediction.**

Case 2: Parameters → $w_1 = 0.1, w_2 = 50, b = 50$

Predicted price:

```
= 0.1 × 2000 + 50 × 5 + 50
= 200 + 250 + 50 = 500k → Perfect!
```

→ **Good prediction.**

Observation:

- Large-valued features (like x_1) → **Small weights**
- Small-valued features (like x_2) → **Large weights**

📉 Cost Function Visualized

- x-axis = w_1 (weight for size), y-axis = w_2 (weight for bedrooms)
- Because of the unbalanced feature ranges:
    - The **cost function contour** is **stretched** (elliptical).
    - Small steps in w_1 make **big prediction changes**.
    - Steps in w_2 hardly change predictions at all.

So gradient descent **struggles to move efficiently**.

🚀 Solution: Feature Scaling

✅ What is Feature Scaling?

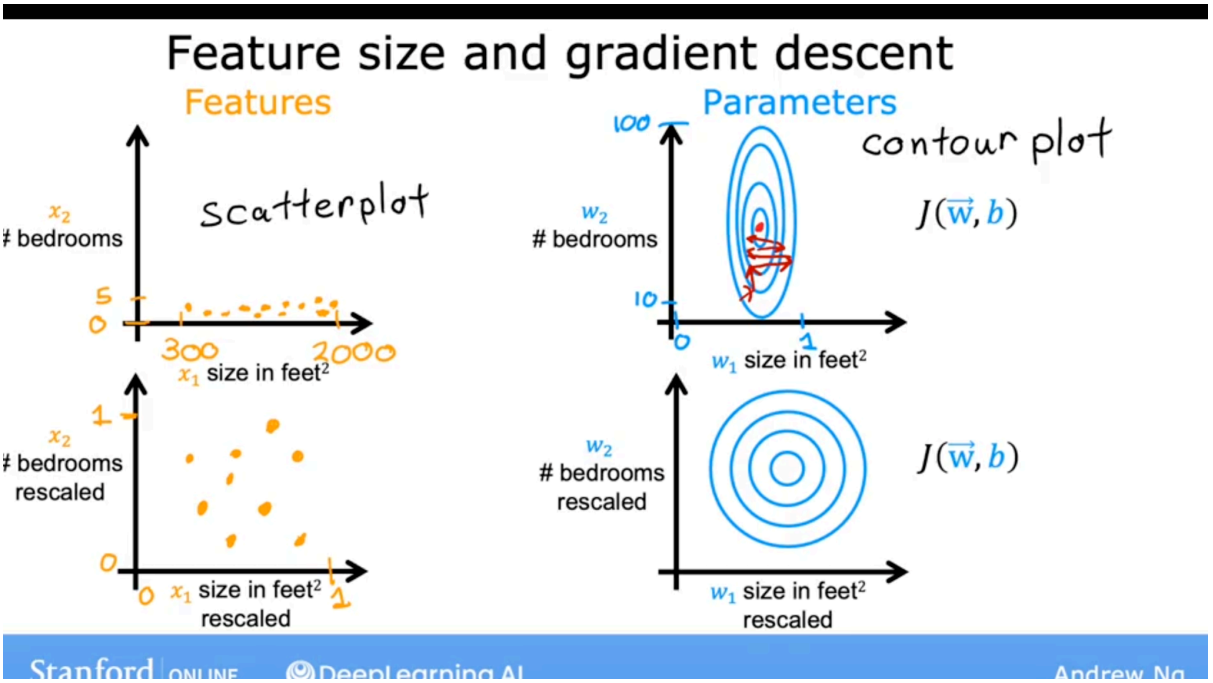
Rescale features so they all take **similar ranges** (e.g., from 0 to 1).

Example:

Feature	Original Range	After Scaling
x_1 (size)	300 to 2000	0 to 1
x_2 (bedrooms)	0 to 5	0 to 1

Now:

- Cost function contours are **circular**.
- Gradient descent moves **directly toward the minimum**.
- **Convergence is faster and more stable.**



📏 Feature Scaling: How to Implement It

When your features (input variables) have very different **ranges**, gradient descent can become inefficient. Feature scaling solves this.

🛠️ Method 1: Min-Max Scaling (Rescaling to [0,1])

🔢 Formula:

$$x_{\text{scaled}} = \frac{x}{x_{\max}}$$

✅ Example:

- Feature x_1 = house size $\in$ [300, 2000]:

$$x_1^{\text{scaled}} = \frac{x_1}{2000} \Rightarrow x_1^{\text{scaled}} \in [0.15, 1]$$

- Feature x_2 = number of bedrooms $\in$ [0, 5]:

$$x_2^{\text{scaled}} = \frac{x_2}{5} \Rightarrow x_2^{\text{scaled}} \in [0, 1]$$

🛠️ Method 2: Mean Normalization (Center around 0)

🔢 Formula:

$$x_{\text{normalized}} = \frac{x - \mu}{x_{\max} - x_{\min}}$$

- μ: mean of the feature
- $x_{\max}, x_{\min}$: max and min values in dataset

✅ Example:

- Suppose:
  - $\mu_1 = 600$, $x_1 \in [300, 2000]$

$$x_1^{\text{normalized}} = \frac{x_1 - 600}{2000 - 300} \Rightarrow x_1^{\text{normalized}} \in [-0.18, 0.82]$$

- For x_2, if $\mu_2 = 2.3$, $x_2 \in [0, 5]$:

$$x_2^{\text{normalized}} = \frac{x_2 - 2.3}{5} \Rightarrow x_2^{\text{normalized}} \in [-0.46, 0.54]$$

🛠️ Method 3: Z-Score Normalization (Standardization)

🔢 **Formula:**

$$x_{\text{z-score}} = \frac{x - \mu}{\sigma}$$

- μ: mean
- σ: standard deviation

✅ **Example:**

- Feature 1: $\mu = 600, \sigma = 450$

$$x_1^z = \frac{x_1 - 600}{450} \Rightarrow x_1^z \in [-0.67, 3.1]$$

- Feature 2: $\mu = 2.3, \sigma = 1.4$

$$x_2^z = \frac{x_2 - 2.3}{1.4} \Rightarrow x_2^z \in [-1.6, 1.9]$$

✅ Rule of Thumb

You ideally want **all features to lie between:**

- **−1 to +1**
- Or loosely within **[−3, +3]** is acceptable
- Avoid ranges like [0, 100] or [−100, 100]

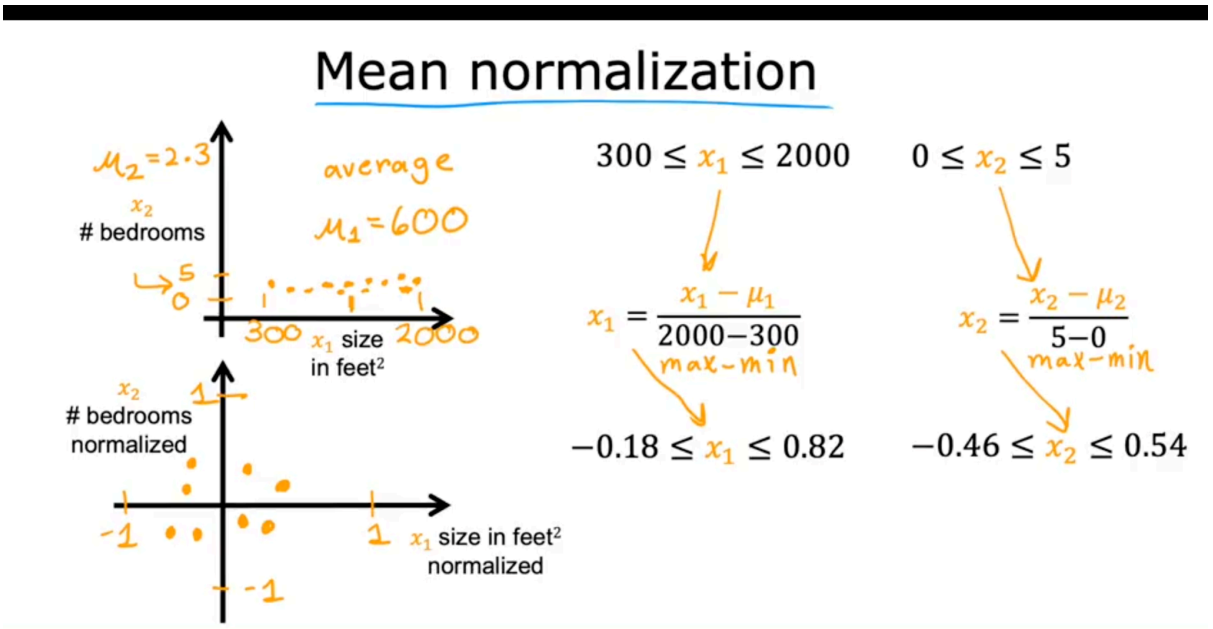
🚩 When to Scale?

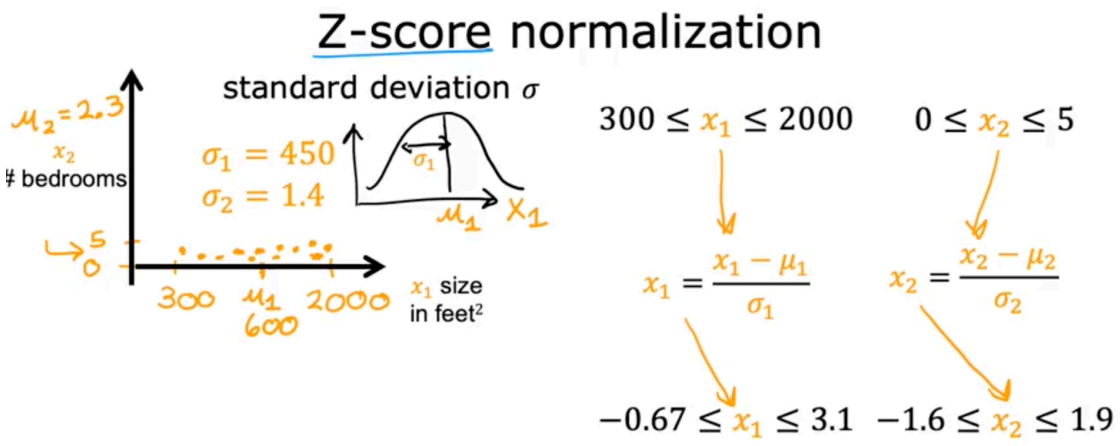
Feature Range	Scale ?	Why?
[0.001, 0.01]	✅	Too small → weak gradient signal
[−100, 100]	✅	Too large → zig-zag descent
[−1, 1]	❌	Already in ideal range
[0, 3]	❌	Acceptable
[98.6, 105] (body temp in °F)	✅	Around 100 → can slow down GD

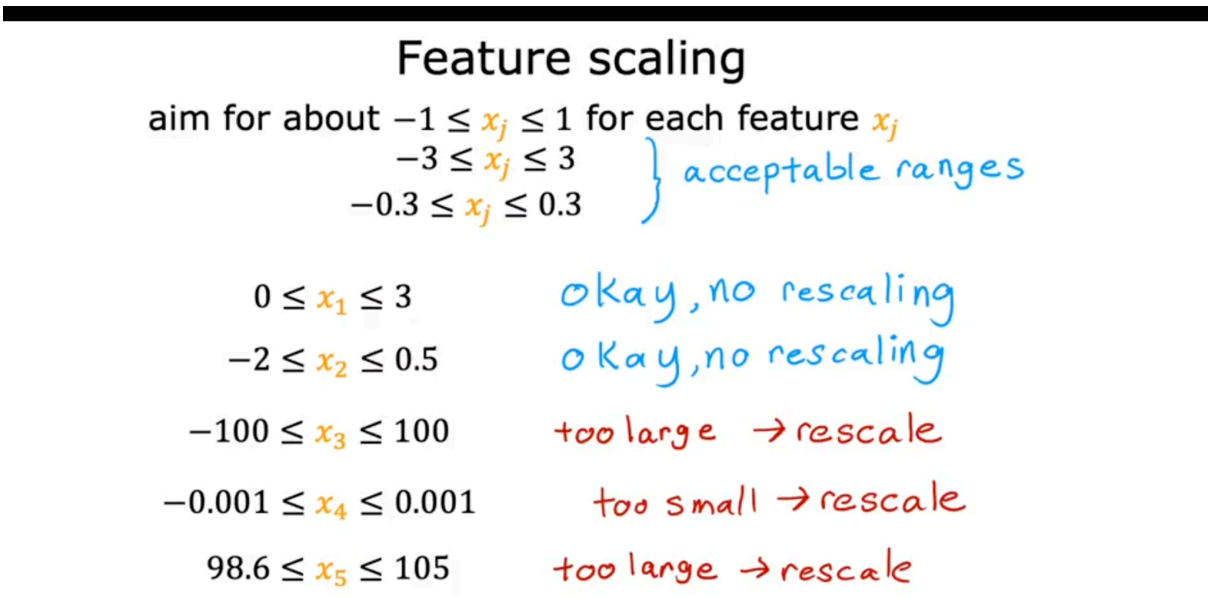
🧪 Want Code for It?

Here's a quick example in Python using NumPy:

```
import numpy as np

# Example feature
x1 = np.array([300, 600, 1000, 1500, 2000])

# 1. Min-Max Scaling
x1_minmax = x1 / np.max(x1)

# 2. Mean Normalization
mu1 = np.mean(x1)
x1_meannorm = (x1 - mu1) / (np.max(x1) - np.min(x1))

# 3. Z-score Normalization
sigma1 = np.std(x1)
x1_zscore = (x1 - mu1) / sigma1
```







📉 How to Know If Gradient Descent Is Converging?

The **goal of gradient descent** is to find parameters www and bbb that **minimize the cost function** J(w,b)J(w, b)J(w,b). So we want to monitor if JJJ is **decreasing steadily** over time.

📊 Step 1: Plot the Learning Curve

✅ What is a learning curve?

- **X-axis**: Number of gradient descent iterations
- **Y-axis**: Value of cost function J(w,b)J(w, b)J(w,b)

📌 What should it look like?

- The curve should **decrease after every iteration**
- It should **flatten out** over time (that's convergence)

🧠 Interpretation:

Observation	Meaning
J decreases steadily	Gradient descent is working well ✅
J increases at some iteration	Learning rate might be too large ⚠️ or code has a bug ❌
J flattens out	Gradient descent has **converged** 🎯
J decreases too slowly	Learning rate might be too small 🐌

🧪 Step 2: Use Epsilon (ε) Convergence Test

Let's define:

$$\varepsilon = \text{a small threshold, e.g., } 0.001 \text{ or } 10^{-3}$$

If the **change in cost function** between two iterations is smaller than ε:

$$|J^{(i)} - J^{(i+1)}| < \varepsilon$$

You can consider that **convergence has happened**.

⚠️ Caution:

Choosing the right value for epsilon can be **tricky**. If it's too small, convergence may never be detected.
That's why:

> **Visualizing the learning curve** is often more reliable than automatic convergence checks.

🧰 How Many Iterations Are Enough?

There's **no fixed number** — it depends on:

- The learning rate α\alphaα

- The complexity of data
- The cost function shape

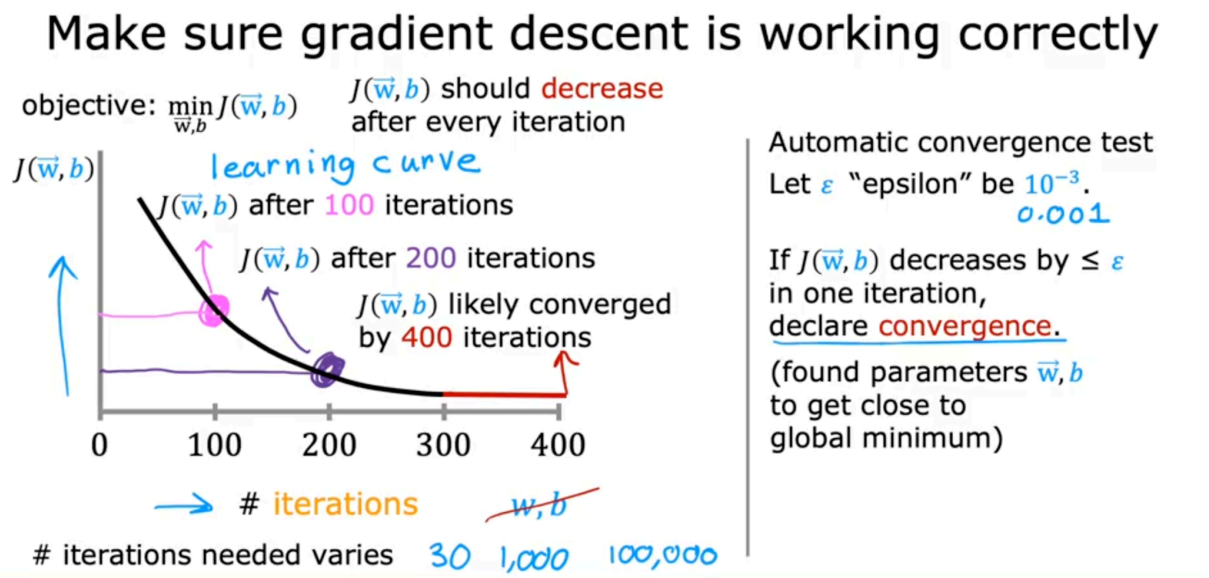
💡 Sometimes 30 iterations are enough. Other times, you may need **100,000+**.

So always **plot the learning curve** — that's your best guide.

📉 Example of a Good Learning Curve

```
Iteration      Cost J(w, b)
---------      ------------
0              120
10             75
50             31
100            17
150            9
200            4.2
250            3.9
300            3.8 (almost flat now)
```

This shows **fast convergence** and then **flattening out** — ideal!



⚙️ Choosing a Good Learning Rate (α)

🎯 Goal of Learning Rate

The learning rate α\alphaα controls how big each step is when updating your model parameters www and bbb during gradient descent:

w:=w−α⋅∂J∂ww := w - \alpha \cdot \frac{\partial J}{\partial w}w:=w−α⋅∂w∂J b:=b−α⋅∂J∂bb := b - \alpha \cdot \frac{\partial J}{\partial b}b:=b−α⋅∂b∂J

❌ What Happens If α is Too Large?

- Updates **overshoot the minimum**
- Cost function JJJ may **bounce up and down**
- Gradient descent may **diverge**, i.e., J increases after each step
- May look like this on the learning curve:

makefile
CopyEdit

```
J:  50 → 70 → 40 → 80 → 30 → 💥 (oscillates wildly or explodes)
```

🧨 Common Mistake in Code:

Wrong update sign:

```
# Wrong ❌
w = w + α * gradient
```

Should be:

```
# Correct ✅
w = w - α * gradient
```

> The minus sign is **critical** because it moves you **toward the minimum**.

🐢 What Happens If α is Too Small?

- Gradient descent works… but **very slowly**
- Takes **forever to converge**
- Example learning curve:

```
J: 50 → 49.9 → 49.8 → 49.7 → ... (tiny progress)
```

✅ What Should a Good Learning Curve Look Like?

- **Cost J decreases rapidly at first**, then **flattens**
- No upward spikes
- Example:

```
J: 50 → 32 → 21 → 15 → 8 → 4 → 3.8 → 3.7 ...
```

🧪 Debug Tip (Powerful!)

> **Try a very small α** (like 0.0001) and see if J decreases every iteration.

If it **still increases**, you likely have a **bug in your gradient descent code**.

🧠 Strategy for Choosing α

1. Try a range of α values:

Start with small values and **increase by a factor of ~3**:

0.001, 0.003, 0.01, 0.03, 0.1, 0.3, 1.00.001, \ 0.003, \ 0.01, \ 0.03, \ 0.1, \ 0.3, \ 1.00.001, 0.003, 0.01, 0.03, 0.1, 0.3, 1.0

2. For each α:

- Run **just a few iterations** (say 100)
- Plot cost JJJ vs iterations
- Pick the **largest α** that **still causes smooth decrease in J**

> You want to pick **the fastest learning rate that still behaves well**.

🔁 Recap: How to Choose α

Step	Action
✅ 1	Try values like 0.001, 0.003, 0.01, 0.03, 0.1
✅ 2	Plot cost J for each α
✅ 3	If J goes up or oscillates, α is too large
✅ 4	If J decreases very slowly, α is too small
✅ 5	Choose the **largest α** that still gives a **smooth descent**

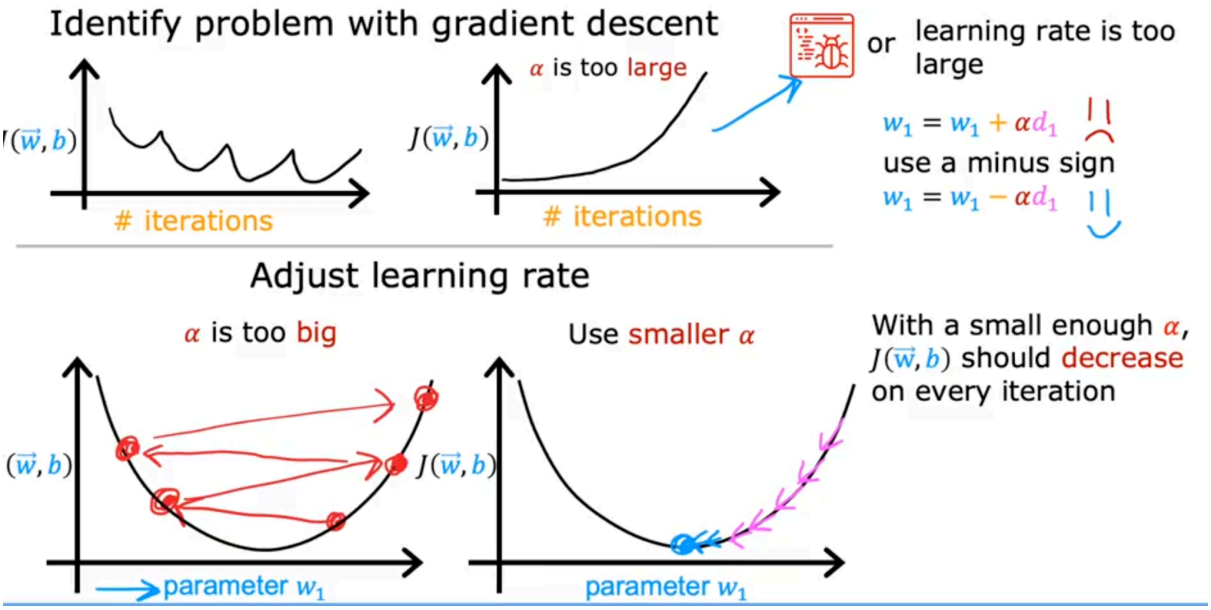
🔬 Real-World Practice

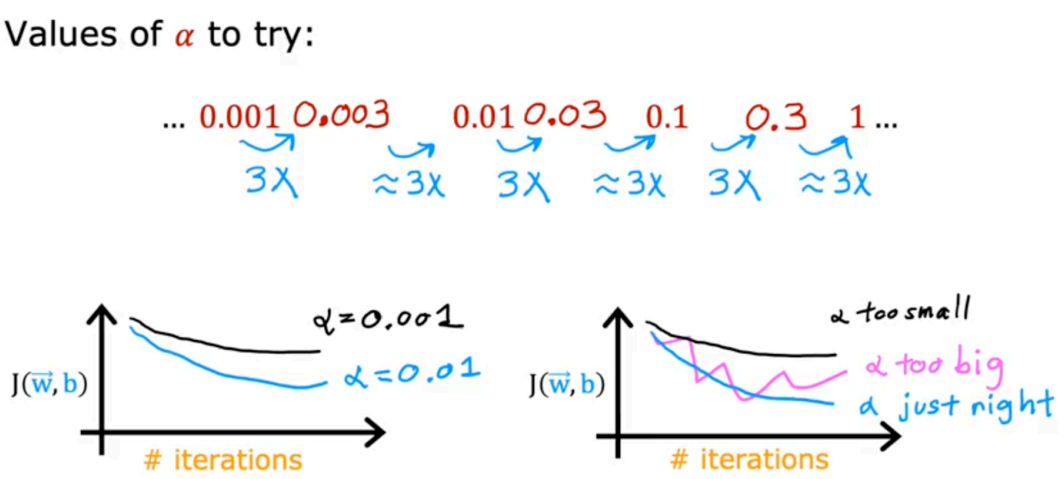
In code, your process might look like:

```
alphas = [0.001, 0.003, 0.01, 0.03, 0.1]
for alpha in alphas:
    print(f"Testing alpha = {alpha}")
    w, b = initialize_parameters()
    J_history = gradient_descent(X, y, w, b, alpha, iterations=100)
    plot(J_history)
```

💬 Final Insight

> **Tuning learning rate is a mix of science and intuition**. Don't expect to find the perfect α instantly. Use plots, play around, and trust your eyes.





You run gradient descent for 15 iterations with $\alpha = 0.3$ and compute $J(w)$ after each iteration. You find that the value of $J(w)$ increases over time. How do you think you should adjust the learning rate α?

- ○ Try running it for only 10 iterations so $J(w)$ doesn't increase as much.
- ○ Keep running it for additional iterations
- ◉ Try a smaller value of α (say $\alpha = 0.1$).
- ○ Try a larger value of α (say $\alpha = 1.0$).

✓ Correct
Since the cost function is increasing, we know that gradient descent is diverging, so we need a lower learning rate.

💡 What is Feature Engineering?

Feature Engineering is the process of:

- **Creating new features** from the raw data you already have.
- **Transforming or combining features** in a way that makes your model perform better.
- Using **intuition or domain knowledge** to guide how you represent input data to the algorithm.

🏠 Example: Predicting House Prices

You have 2 original features:

- x_1: **Frontage** – width of the lot (in feet or meters)
- x_2: **Depth** – length of the lot

A basic linear model would be:

$$f(x) = w_1 \cdot x_1 + w_2 \cdot x_2 + b$$

This uses **frontage and depth as independent predictors**.

But here's the smart observation:

> 🤯 **Area = Frontage × Depth = $x_1 \times x_2$**

So we engineer a **new feature**:

- $x_3 = x_1 \cdot x_2$ → **Area of the land**

Now, you build a better model:

$$f(x) = w_1 \cdot x_1 + w_2 \cdot x_2 + w_3 \cdot x_3 + b$$

The model will **learn** whether width, depth, or area (or some combination) is most predictive of house price.

🔧 Why Feature Engineering Matters

- It **gives the algorithm better signals**.
- It helps capture **non-linear patterns** without needing fancy models.
- It can **improve accuracy significantly**, especially when raw features are not sufficient.

🧠 Key Insight

> Rather than blindly using the features given to you, think about the **real-world relationships** between them. You might be able to **combine or transform** them into more powerful predictors.

Feature engineering

$$f_{\vec{w},b}(\vec{x}) = w_1 x_1 + w_2 x_2 + b$$

frontage depth

$$area = frontage \times depth$$

$$x_3 = x_1 x_2$$

new feature

$$f_{\vec{w},b}(\vec{x}) = w_1 x_1 + w_2 x_2 + w_3 x_3 + b$$

Feature engineering: Using intuition to design new features, by transforming or combining original features.

If you have measurements for the dimensions of a swimming pool (length, width, height), which of the following two would be a more useful engineered feature?

- (●) $length \times width \times height$
- ( ) $length + width + height$

✓ Correct
The volume of the swimming pool could be a useful feature to use. This is the more useful engineered feature of the two.

📈 Why Polynomial Regression?

Sometimes, your data can't be captured well with a **straight line** (like in linear regression). For example:

> You have data where x is house size in square feet, and y is price.

Plotting this may show a **curve**, not a straight line. So, what do we do?

🔁 From Linear to Polynomial Regression

We **expand the feature set** by including powers of x, like:

- xxx
- x2x^2x2
- x3x^3x3
- … up to xdx^dxd (for degree-d polynomial)

Example:

Example:

Quadratic Regression:

$$f(x) = w_1 \cdot x + w_2 \cdot x^2 + b$$

This fits a **U-shaped curve**. But in housing prices, this might not make sense, because it eventually **comes back down**, which implies that very large houses become **cheaper**, which is unrealistic.

🔺 Cubic Regression:

$$f(x) = w_1 \cdot x + w_2 \cdot x^2 + w_3 \cdot x^3 + b$$

Now you get a **flexible curve** that can better match increasing prices as house size increases.

⚠️ The Importance of Feature Scaling

As you raise x to higher powers:

- If x ranges from 1 to 1,000

- Then x2x^2x2 ranges from 1 to **1,000,000**
- And x3x^3x3 ranges from 1 to **1,000,000,000**

These huge differences in scale can **slow down or break gradient descent**. That's why **feature scaling (e.g., normalization)** is essential.

🧪 Other Feature Choices

Polynomial regression doesn't just mean powers like x^2 or x^3. You can get creative:

- Use $\sqrt{x}$ (square root of x)
- Use $\log(x)$
- Combine features in new ways

Example:

$$f(x) = w_1 \cdot x + w_2 \cdot \sqrt{x} + b$$

The square root curve **increases more slowly** as `x` grows and **never comes down**, which might be a better fit for real-world housing price trends.

💡 Feature Engineering + Polynomial Regression = Power

You're **engineering features** (transforming `x` into $x^2, x^3, \sqrt{x}$, etc.) to help the model **understand the pattern better**.

Instead of just:

$$f(x) = w_1 \cdot x + b$$

You now have:

$$f(x) = w_1 \cdot x + w_2 \cdot x^2 + w_3 \cdot x^3 + b$$

Or:

$$f(x) = w_1 \cdot x + w_2 \cdot \sqrt{x} + b$$

The model can now fit **non-linear relationships**.

Polynomial regression